Analysis of LIS Algorithms and Blockchain Transaction Validation: Precision and Performance Insights

Niccolò Mascaro
n.mascaro@gmx.com

October 25, 2024

Abstract

This paper presents a comparative analysis of two approaches to solving the Longest Increasing Subsequence (LIS) problem: the traditional dynamic programming method with a time complexity of $O(n^2)$ and an optimized method employing binary search with a time complexity of $O(n \log n)$. Through extensive experimentation on various types of input sequences, the study evaluates both performance metrics and the nature of the subsequences generated by each algorithm. The results demonstrate significant execution time improvements in the optimized approach and reveal that different algorithms may produce distinct yet valid LIS results under certain conditions. These insights are particularly relevant for applications requiring both efficiency and consistency, such as blockchain transaction validation and other real-world systems. The results of this study have been published in a public repository, which can be accessed at `https://github.com/mascarock/lis-study`.

1 Introduction

The Longest Increasing Subsequence (LIS) problem is a fundamental challenge in computer science and combinatorics. Given an array of integers, the objective is to identify the longest subsequence where the elements are strictly increasing. The LIS problem has applications in various domains, including bioinformatics, pattern recognition, financial analytics, and machine learning.

For example, consider the following series of numbers:

> Remove the fewest possible numbers such that the sequence remains in increasing order: [9, 44, 32, 12, 7, 45, 31, 98, 35, 41, 8, 20, 27, 32, 83, 64, 61, 28, 39, 93, 29, 92, 17].
>
> In this case, one possible longest increasing subsequence is: [9, 12, 31, 35, 41, 83, 93].

This study provides a comprehensive analysis of two prevalent LIS algorithms, evaluating their performance and the characteristics of their outputs across different input scenarios.

2 Methods

The algorithms discussed in this paper have been implemented in Python. Below are the implementations for both approaches used in our analysis.

2.1 Dynamic Programming Approach — $O(n^2)$

The following Python code implements the traditional dynamic programming approach [1]:

```
# First approach - O(n^2)
def find_longest_increasing_subsequence(arr):
    n = len(arr)
    # dp[i] stores the length of the longest subsequence
        ↪ ending at arr[i]
    dp = [1] * n
    # prev[i] stores the index of the previous element in
        ↪ the subsequence
    prev = [-1] * n

    # Find the length of the longest subsequence
    for i in range(1, n):
        for j in range(i):
            if arr[i] > arr[j] and dp[i] < dp[j] + 1:
                dp[i] = dp[j] + 1
                prev[i] = j

    # Find the index of the last element of the longest
        ↪ subsequence
    max_length = max(dp)
    max_index = dp.index(max_length)

    # Reconstruct the subsequence
    result = []
    while max_index != -1:
        result.append(arr[max_index])
        max_index = prev[max_index]

    return result[::-1]  # Reverse the list to get the
        ↪ correct order

# Example usage
numbers = [9, 44, 32, 12, 7, 45, 31, 98, 35, 41, 8, 20, 27,
    ↪ 32, 83, 64, 61, 28, 39, 93, 29, 92, 17]
result = find_longest_increasing_subsequence(numbers)
```

```
print("Longest␣Increasing␣Subsequence:", result)
```

2.2 Dynamic Programming with Binary Search — $O(n \log n)$

The following Python code implements the optimized approach using binary search [1]:

```
import logging
from bisect import bisect_left

# Second approach - O(n log n)
def find_longest_increasing_subsequence_optimized(arr):
    n = len(arr)
    if n == 0:
        logging.info("The␣input␣list␣is␣empty.")
        return []

    # 'tails' to store the smallest tail of all increasing
        ↪ subsequences with length i+1
    tails = []
    prev = [-1] * n
    indices = []

    for i in range(n):
        pos = bisect_left(tails, arr[i])
        if pos == len(tails):
            tails.append(arr[i])
            indices.append(i)
        else:
            tails[pos] = arr[i]
            indices[pos] = i

        prev[i] = indices[pos - 1] if pos > 0 else -1

    # Reconstruct the LIS
    result = []
    k = indices[len(tails) - 1]
    while k >= 0:
        result.append(arr[k])
        k = prev[k]

    return result[::-1]

# Example usage
numbers = [9, 44, 32, 12, 7, 45, 31, 98, 35, 41, 8, 20, 27,
    ↪ 32, 83, 64, 61, 28, 39, 93, 29, 92, 17]
result_optimized =
    ↪ find_longest_increasing_subsequence_optimized(numbers)
print("Longest␣Increasing␣Subsequence␣(Optimized):",
    ↪ result_optimized)
```

3 Results

The experiments were conducted on various types of input lists to evaluate both the performance and the qualitative differences in the Longest Increasing Subsequences generated by each algorithm. The input types included:

- **Random List:** The input list contains random values.
- **Sorted List:** The input list is sorted in ascending order.
- **Reversed List:** The input list is sorted in descending order.
- **Constant Values:** The input list has all identical values.
- **Alternating Peaks:** The input alternates between low and high values.
- **Zigzag:** The input list rises and falls in a zigzag pattern.

3.1 Performance Metrics

Figure 1 illustrates the execution time of both algorithms across varying input sizes. As anticipated, the optimized $O(n \log n)$ approach significantly outperforms the traditional $O(n^2)$ method, especially as the input size increases. This performance gain underscores the practicality of the optimized approach for large-scale applications.

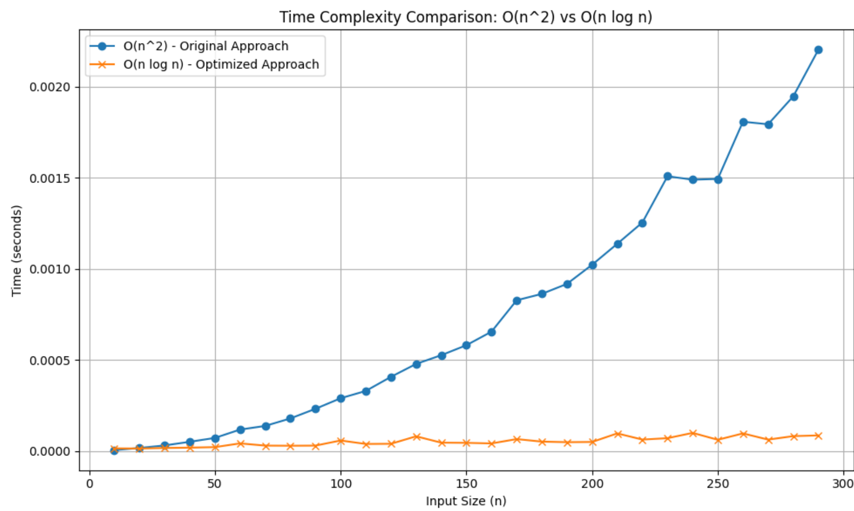

Figure 1: Time Complexity Comparison: $O(n^2)$ vs. $O(n \log n)$

3.2 Qualitative Analysis of LIS Outputs

The experiments reveal that different algorithms may produce different LIS results even when the length of the subsequences is identical. The following observations were made across various input types:

- **Random List:**
  - Optimized algorithm output: [1, 13, 21, 26, 61, 93]
  - $O(n^2)$ approach output: [41, 54, 69, 75, 89, 93]
- **Sorted List:** Both algorithms correctly identify the entire list as the LIS.
- **Reversed List:** Both algorithms determine that the longest increasing subsequence is a single element, though different elements may be selected.
- **Constant Values:** Both algorithms identify that the LIS consists of a single repeated value.
- **Alternating Peaks:** Both algorithms successfully identify the same LIS, demonstrating robustness in fluctuating input patterns.
- **Zigzag:**
  - Optimized algorithm output: LIS with small increments.
  - $O(n^2)$ approach output: Slightly different subsequence due to iterative nature.

These variations indicate that while both algorithms are capable of identifying a valid LIS, their internal mechanisms lead to different valid outputs under certain conditions. Specifically, the optimized $O(n \log n)$ method tends to select smaller values earlier in the sequence, potentially resulting in different LIS compared to the $O(n^2)$ approach.

4 Discussion

The Longest Increasing Subsequence (LIS) problem can be addressed using various algorithmic strategies, each presenting distinct trade-offs in terms of time complexity and the nature of the subsequences produced. This study compares the traditional $O(n^2)$ dynamic programming approach with an optimized $O(n \log n)$ method that incorporates binary search.

4.1 Dynamic Programming Approach

The traditional dynamic programming approach meticulously evaluates all possible subsequences to determine the longest increasing one. By maintaining a `dp[]` array where each element `dp[i]` represents the length of the LIS ending at position i, the algorithm ensures that all potential subsequences are considered [1]. However, this exhaustive evaluation results in a quadratic time complexity, making it less efficient for large input sizes.

4.2 Dynamic Programming with Binary Search

The optimized approach leverages binary search to maintain an auxiliary `tails[]` array, where each element signifies the smallest possible tail of an increasing subsequence of a given length [1]. This method significantly reduces the time complexity to $O(n \log n)$, enhancing performance for larger datasets. However, the experiments indicate that while this approach is more efficient, it may produce different valid LIS results compared to the traditional method due to its tendency to choose smaller elements earlier in the sequence.

4.3 Implications of Different LIS Outputs

The ability of different algorithms to produce distinct LIS results of the same length has practical implications:

- **Consistency in Applications:** In systems where consistency of the LIS output is crucial—such as blockchain transaction validation—choosing an algorithm that provides deterministic results is important. The $O(n^2)$ approach, with its exhaustive search, may offer more predictable subsequence generation compared to the optimized method.
- **Efficiency vs. Predictability:** While the optimized $O(n \log n)$ approach excels in efficiency, especially with large inputs, it may not always provide the same subsequence as the traditional method. This trade-off must be considered based on the application's requirements.
- **Handling Multiple LIS:** Given that multiple LIS of the same length can exist for a single input, algorithms differ in their selection process. The optimized approach's preference for smaller elements can be advantageous or detrimental depending on the context [4].

4.4 Applications in Blockchain Environments

Integrating LIS algorithms within blockchain frameworks, particularly in Ethereum Virtual Machine (EVM) environments, presents opportunities to enhance transaction sequence validation and system efficiency.

Optimizing Transaction Sequence Validation: Implementing the optimized $O(n \log n)$ LIS algorithm can streamline the validation of transaction sequences in smart contracts. For example, in decentralized exchanges (DEXs), ensuring transactions are processed in a logically increasing order based on timestamps or IDs can be efficiently managed using LIS, reducing computational overhead and gas consumption.

Enhancing Scalability and Efficiency: As blockchain networks scale, the demand for efficient algorithms becomes paramount. The optimized LIS approach can contribute to scalable solutions by minimizing the number of comparisons and leveraging its logarithmic time complexity, thereby facilitating faster transaction processing and lower operational costs.

Fraud Detection and Security Enhancements: Analyzing transaction patterns with LIS can aid in identifying irregularities indicative of fraudulent activities. This proactive approach can strengthen the security mechanisms of blockchain networks, fostering greater trust and reliability in decentralized applications.

Real-World Implementation and Testing: Future research should focus on the practical implementation of LIS algorithms within existing blockchain platforms. Empirical studies and performance evaluations in real-world EVM environments will provide insights into the benefits and challenges of such integrations, guiding the development of optimized smart contracts that leverage LIS for enhanced performance and security.

5 Conclusion

This analysis confirms that the optimized $O(n \log n)$ approach is more suitable for solving the LIS problem, particularly for larger input sizes, due to its superior execution time. However, the study also highlights that different algorithms may produce distinct yet valid LIS results, emphasizing the importance of selecting the appropriate method based on application-specific requirements for consistency and efficiency.

6 Further Research

While this paper provides a comprehensive analysis and optimization of algorithms for solving the Longest Increasing Subsequence (LIS) problem, several avenues remain open for future investigation. Exploring these areas can enhance the applicability and efficiency of LIS algorithms in various domains, including emerging technologies such as blockchain.

6.1 Exploration of LIS in Other Emerging Technologies

Beyond blockchain, the LIS algorithm holds potential applications in other cutting-edge fields such as artificial intelligence, bioinformatics, and financial analytics. Investigating these applications can lead to innovative solutions that leverage the algorithm's efficiency and versatility.

6.2 Algorithmic Enhancements and Hybrid Approaches

Further research could explore hybrid approaches that combine LIS with other optimization techniques. Integrating machine learning models or parallel processing strategies with LIS algorithms may yield even greater performance improvements, opening new possibilities for tackling complex computational problems.

7 References

References

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press.

[2] Knuth, D. E. (1973). *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley.

[3] Wikipedia. (2024). *Longest increasing subsequence*. Retrieved from `https://en.wikipedia.org/wiki/Longest_increasing_subsequence`

[4] CP-Algorithms. (2024). *Longest Increasing Subsequence*. Retrieved from `https://cp-algorithms.com/sequences/longest-increasing-subsequence.html`

[5] Baeldung. (2024). *Longest Increasing Subsequence Using Dynamic Programming*. Retrieved from `https://www.baeldung.com/cs/longest-increasing-subsequence`